Technical Reports: A pedagogical reflection

Software Development 2021
Department of Computer Science
University of Copenhagen

Alexander Christensen — jpb483

June 24, 2021

Contents

Foreword

Report writing is a complicated topic. You could almost call it a discipline in and of itself. There is something soothing about reading a well-written scientific article where every word and sentence melt together as a unified whole; where the intent of the writers is crystal clear, and when any superfluous filling has been left out. You could almost call it a relaxing experience. It's certainly pleasant. And this is especially important when dealing with advanced topics with many complex mathematical formulas and programming concepts at hand: If the language and the grammatical constructs are difficult to comprehend, then the reader has to do double work.

So we know what it is we desire, but how can we achieve it; how does one approach report writing keeping this intent in mind? This guide is designed to ease the understanding of what separates good report writing from bad. We provide a report format that, although quite a common setup, we have customized and adopted for the programming course Software Development at the University of Copenhagen, dep. computer science. We shall, henceforth refer to this format as the *technical report*, and in this writing explain our understanding of this format.

1 Basics

We start with the fundamental aspects of report writing, which are common for any writing style, fiction or non-fiction: Language and grammar.

It is difficult to place a precise mark on correct and incorrect ways of writing, also when writing scientific reports and papers. But there do exist some core guidelines that we will shed some light on:

- Language semantics
- Spelling
- Grammar
- Text flow
- Coherency
- Introductory

We will briefly comment on each of those aspects, without bein too concrete or detailed. A reason for this choice will follow shortly.

Language semantics

Semantics is a commonly used term in programming language design, and refers to the meaning of textual constructs. For example, you might have a certain algorithmical function implemented with a `while`-loop. Now, we imagine that our language of choice allows us to use a functional-style notation as well, so we aptly reinterpret this algorithm as a tail-recursive function. We could say that this implementation follows the same semantics, as the code carries the same meaning albeit looking different.

A bit more analysis on language semantics (ie. not *programming* language semantics) presents us with a concept called "compositional semantics" which is very accurately described as this:

The principle of compositionality asserts that the meaning of an expression is composed of the meaning of its parts and how the parts are combined structurally[1]

This statement may be applied on a multiple of domain areas, so we will let you ponder on its philosophical meaning for a moment.

Spelling and grammar

These are more straightforward. It is always recommended to read through your written works one last time before submitting. For many people spelling errors are a thorn in their eyes and lowers their respect for what is written. Spelling can be checked automatically by various tools, while grammer is a bit more complicated to deal with.

Grammer includes placement of comma, semi-colon, and other sentential delimiters. It also includes using correct prepositions such as *"at the University"* (correct) versus *"in the University"* (incorrect), uppercasing of certain words: University, January, Monday, etc., and the ordering of words within a sentence.

[1] `https://scholar.harvard.edu/files/adam/files/meaning.ppt.pdf`

Text flow

As my background in music permits me, my understanding of text flow is that of rhythm and pacing. For any piece of text, we desire *fluency*, that is - the words and sentences to appear in a natural order where they can be read without any trouble, stumbling, or pause. Use of alitterations, triads, punctuation, word complexity, etc., all works towards ensuring good fluency such that the reader has a pleasant experience reading your report.

Coherency

Coherency is more closely related to the contents, and how it fits together on a larger scale, rather than small word-to-word adjustments or grammatical fiddling. When writing your report, think about how each section of text relates to its previous sections or paragraphs, in terms of flow, grammar, semantics, and contents. If a large section of text has been written using past tense, and then you suddenly switch to present tense, then your text might appear incoherent (unless the use of tense closely correlates to the contents).[2] Another example is that of a conclusion writing about stuff which has not been analysed or otherwise explained previously in the report. Also incoherent.

Introductory

And we have ended our tour of writing fundamentals at the beginning: The introduction. Alternatively, in written works such as here you might often find a foreword instead. And while there may be technical differences between these two, they share a common purpuse: Ease the reader into the report.

This is very important, and can be compared to waking up in the morning with your neighbours screaming, the building across the street on fire, or similar vivid imaginaries. In martial arts there is always a mandatory ceremonial warmup routine, often in the form of meditation. The reason behind is so we can enter the task at hand (performing martial arts, reading a report, etc.) with an empty and relaxed state of mind. Two short examples will suffice:

1.1. Our game uses the Strategy design pattern to simulate movement strategies between the different chess pieces.

1.2. Our implementation simulates the game chess, where we have applied the Strategy design pattern to alternate between movement strategies for different game pieces.

Perhaps a bit contrived, but there are some fundamental differences: Example 1 discusses movement strategy, game, and strategy design pattern, before we are even aware that this is a report about the game chess.

Example 2 presents the context, ie. an implementation of the game chess, *then* describes the application of certain design patterns and their usages.

[2] By the way, see what I did there - "Closely Correlates [...] Contents" (CCC).

Ease the reader into your written works through proper introductory. We start the morning with a hot, warm cup of coffee (or Tea, God forbid!), and a relaxing display of morning slumberness in our favorite chair. The more intuitive and relaxing the experience of your introduction is to the reader, the more they will feel motivated and engaged to read the rest of your report.

2 Personal touch

We mentioned earlier that we would not go too much into detail with the fundamentals, and that we would provide a reason for this. Here is our rationale:

It goes for report writing, as for software development as a whole, that we are presented with a variety of tools, and as such should aim to develop our understanding of when and when not to apply these tools. We have a couple of guidelines, as well as a formal report structure that we will look at in the next couple of sections; but they are just guidelines. If we were to tell you exactly how to write every section of a report, we would compromise your opportunity for autonomy and expressed individuality, which would be even worse. So, in order to leave space for the personal touch, and for the unknown, we leave it as an exercise to the reader to figure out the rest. [3]

3 Technical report format

Our *technical report* format has a structure that you **must** follow for the SoftwareDevelopment course. This is designed, and intended, as a pedagogical exercise so that you may develop an intuition of good report design. The structure goes as follows:

3.1. Introduction

3.2. Background

3.3. Analysis

3.4. Design

3.5. Implementation

3.6. Evaluation

3.7. Conclusion

While you may want to customize each section to suit your personal preferences you may not switch the order of sections or omit any of them.

[3] Just kidding. Not really though.

3.1 Introduction

We have already touched upon this. An introducction should explain the context and circumstances of the report. In other words, after reading the introduction the reader must have an understanding of what this project is and why you have made it.

We have an example, a very short one. Please do not copy/paste. We really encourage you to find your own writing style:

In this fourth assignment of the course Programming Language Design, we have been tasked with completing a handed out interpreter for a simple, functional-style programming language called "Fasto". We will explain our design choices along the way, any ambiguities we have found in the assignment text, as well as some areas where we chose to divert from course curriculum and follow our own design style as we found that more intuitive.

Great! Now the reader knows exactly what this report will be about and why we wrote it. And she didn't have to wake up to any screaming neighbours or burn her tongue on heated, hastily-brewed coffee.

3.2 Background

Following the "Fasto" example from before, we will be using the Background section for a more in-depth view of the problem domain. Here we would explain that the interpreter will be built in F#, a short bit of history about "Fasto" (who created it and when), and some summary about what features the finished product should support. If some tasks have been simplified, e.g. for the sake of learning, this would also be an appropriate place for mentioning that.

You may view the Background as an extension to the Introduction. There should be a *fluent* transition into this section, and it should appear *coherent* together with the Introduction.

3.3 Analysis

This section is very tricky to get right, often the one spent the most time on, and definitely the one that supports the most variety. So we will start at the intent, what we desire from a good analysis, and slowly build our way up.

3.3.1 Why do we analyse?

Analysis is a word that is often met with a great deal of confusion. I have heard this question often: *"What are we supposed to analyse? It says right here what we should do."*. And this is very understandable. But we can dispel this confusion with a simple insight: We are writing for the reader, not for ourselves. While it may be easy for us to know what we should do, the reader may not possess the same knowledge. So we should help them as much as we can. And this is especially true when the report is placed in a teaching environment such as a University course: The reader/TA/examinator needs to know that you have understood the requirements of the assignment. So before we are explaining

how we do things, we must explain what we will be doing. That sounds rather intuitive, wouldn't you agree?

3.3.2 What should we not write in an Analysis?

We should not be making any design choices. We should not write how we are going to solve things. That will come later. If we have decided that a specific design pattern would be suitable for some part of the code, this is *not* the place to mention this design pattern or why we use it. Even if the assignment was about finding an appropriate design pattern for a problem, we should just mention that and leave it there. Which design pattern and why will come later.

3.3.3 What is appropriate to write about then?

Stating the problems to be solved. Stating the requirements. Stating any features that we must implement. If we felt that the assignment text was unclear of certain requirements, e.g. due do ambiguity, we will use the Analysis to either make a choice or to invent our own requirements (and write why!).

3.3.4 Should we include a UML-diagram in the Analysis?

No. We should not. UML-diagrams represent concious design choices, and should be postponed to later sections. Exception: If the assignment text presented us with a UML-diagram that we should follow or analyse, in which case we should obviously insert that here (or in appendix if it takes up a lot of space).

3.4 Design

Now that we have presented the problem at hand it's time to explain how we are going to solve it. And we make a note here: There is a significant difference between how we are going to solve it, and how we actually *did* solve it. The latter will include technical details in our programming language, while the former is going to be more high-level. And Design is all about the high-level overview.

3.4.1 Why we have the Design section

It is desireable to be able to understand the application that you have solved without actually looking at the code. This saves a lot of time for whoever is grading your projects. Imagine them having to look through every single code file and analyse their contents. And then imagine them reading a couple of pages in a report. The second is by far the preferred choice.

So you should aim to write a Design section such that the reader may *efficiently* get an overview of your entire application - not the requirements or features (those were explained in the Analysis), but how you have *chosen* to implement them.

3.4.2 What should we not write about?

We should not insert code snippets, unless they sketch out a particular algorithm that we have chosen as part of our design. We should also not state requirements for the application or list a couple of features - we have the Analysis for that.

We are also not interested in details such as variable initialization or the lifetime of certain objects in code - those are implementation details, and not part of sketching out an application design.

3.4.3 What should we write about?

You should include everything necessary for the reader to obtain a high-level overview of how you have designed (but not programmed!) your application. This may include UML-diagrams, sequence diagrams, how you have chosen to separate concerns or structured similar or related responsibilities into logical units such as files, classes, or namespaces.

As as example, you may have used a responsibility and associativity analysis earlier, so naturally the next step would be to use this section to describe the (conceptual) conversion from requirements to logical units that can be converted into code. Remember, the Design section should be about a *high-level* overview. A good parallel could be a customer having a meeting with a production company, where the customer presents a series of features that a certain product should facilitate. The company would sketch out all these features and requirements in the Analysis section until all ambiguities had been resolved. The Design section would then be developed by the lead engineers, and handed out to the technical team so the development could start. Up until this point (Design section) there has been no development taking place. [4]

3.5 Implementation

Finally, we have arrived at the central part of the report: The Implementation. This section should take up more space than the other sections, and should focus solely on *non-trivial, low-level* implementation details. So naturally, we will talk a bit about what we mean by that.

3.5.1 Triviality

Some implementation details are more important, or more significant than others. If we have a class which contains an array of elements, then it is the natural choice to add a counter and initialize it to zero. This initialization is a *trivial* implementation detail, or of low *complexity*, so we should not bother the reader with such a minor detail. On the other hand, if we are programming a miniature library to handle grammatical errors in a text file format, then the reader would likely be interested in what kind of errors that may occur and how we

[4] In software development methodologies this approach to project management bares the most resemblance to the Waterfall method, which is in general discouraged for most development projects. In reality, an agile approach will be more suitable.

handle them, ie. can we classify errors into categories of recoverable and unrecoverable?

Or if part of the specification was to implement a state machine for a game, we should explain how we switch between states, in what order states are initialized. If we e.g. switch to `GameRunning` state, then to `MainMenu`, and then again to `GameRunning`: How do we make sure that the state is reset so the user may experience a new game being started?

To aid in your understanding of *triviality*, we have composed a short list of examples that you may reason about on your own:[5]

3.1. Each time a new game is started we set the score counter to 0.

3.2. Our level loader is programmed to detect errors in the level file format, such as images not being present on harddisk, and will in such cases display a solid purple rectangle for visual recognition, and write a text message in the console to aid debugging.

3.3. During our analysis we found no clear distinction between when a game state should be *created* and when it should be *initialized*, so we have chosen to create and initialize all game states in the *StateMachine* constructor.

3.5.2 What do we mean by low-level?

Software and programming is, in general, a huge layer of abstractions built upon each other, supported by decades of research to enable us to perform tasks of high complexity with very few instructions. When we write `var a = new Car();` we expect our programming language to handle details for us such as memory allocation and alignment, returning a pointer, understand what "Car" means - perhaps it is a class declared in another file, converting this into machine code that our CPU can execute, protection against unwanted access such that another program might not suddenly overwrite data on our car by accident; or a wide array of other conspicuous errors. But we don't want to bother by all these things when we write our code, and thus we have the concept of *abstraction*, where we can delve more into details of how things work (more "low-level") or look at things from a more abstract point of view (more "high-level").

In general, we desire the Implementation section to explain *low-level* details for us, so the reader knows that you understand your application, but also not *too* low-level so we lose focus on what's important.

3.5.3 What should we not write about?

We should not conclude on how the application works. We should not describe test results, benchmarks, results from profiling, or other such things. We should also not discuss limitations of our implementation, bottlenecks, or things that didn't go as planned.

[5] If you are still in doubt, then the first example is trivial, the other two are not.

3.6 Evaluation

Evaluating a piece of software is just as important, if not more so, than developing it. We can write as much code as we want, but if we cannot be certain that it works, we will not be able to (ie. should refrain from doing) release it to the public. But testing is not an easy thing. Even if we have perfect path coverage, there are things that we might not know for sure, such as conversion between *string* and *int*, if our data types are appropriate, if our design is "good", or if our program runs optimally.

And as such, we list several topics which a good Evaluation section *must* touch upon.

3.6.1 Playtesting

For the SoftwareDevelopment course we place a focus on game development, and as such you must "playtest" your application. Does it work? Are there visual bugs, glitches, weird rendering artifacts? Are the game mechanics balanced? None of these items can be covered by unit tests, and they are all essential for ensuring a quality product.

3.6.2 Testing

You should explain how to run the tests, how the tests are structured, to what extent your code is tested / not tested. It is also relevant to discuss what kind of testing your have performed: Unit testing, black/white-box, integration testing, acceptance testing, randomized testing, etc.

3.6.3 Issues

If you enountered any issues while developing or testing that have not been fixed, they should also be explained as part of the Evaluation. This could be known bugs, benchmarking results or results from profiling, parts of the code that run unoptimally, if you suspect memory leaks, if the same image is being read from harddisk multiple times, or other things that you would fix if time permits. We consider honesty a display of the highest academic level.

3.7 Conclusion

This section practically writes itself, if all other sections have been completed. At the termination of a project, after the entire report is finished, a Conclusion can be written in 5-10 minutes. It is necessary to explain what the reader already finds evident or suspects: If you failed or succeeded in the development project. It may be as simple as stating the results from evaluating your application, or it may include a short discussion or reflection about the group work, the time frame, shortcomings in the assignment, or any challenges that you faced during development.

It is always good custom to end in such a way that the reader feels the case is closed, that no important details have been left out. She will sip the last coffee, close her eyes, and thank you for a pleasant reading experience.

4 Self-help list

Here you can find questions to common answers, as well as some general advice or things we found significant to include in this guide.

4.1 How to include code

Code snippets should be placed in the Implementation section, or in appendix if they are longer than half a page. You should **never** use screenshots for code. It looks unprofessional, doesn't scale well to different resolutions, and they cannot be copy/pasted. In Latex there are a few different options: `verbatim`, `alltt` (you must include a package for that), `listings`, `Pygments`. You may use a custom set of macros to color your code snippets, but that is not necessary.

4.2 How to include diagram

Diagrams may be included as screenshots. You can use various Latex packages as well, but that is usually quite bothersome. Remember to add a caption to the image describing what is shows. If you don't include a caption the image adds no real value to your report.

4.3 What mindset should I adopt

Granted, very few people will be asking themselves this question. But it is important to ask nontheless. We should aim to write *efficiently*, with as few words as possible. The keywords are *clarity* and *brevity*. It does not mean that we should "cram in" as much information as we can, but it does mean that we should not be overly verbose in our descriptions. By using few words and explain only the neccessary parts we can show the reader that we respect their time. So we should adopt the mindset of *efficiency*.